CHAPTER 3 TEXTURES

Contents

- **Texture mapping**
- Texel vs Pixel
- The basics of texturing
- Texture repeating
- Texture minification
- Texture magnification
- Enter the mini-me of textures
- **Mipmaps**
- Mipmap texture having half linear dimensions
- Determining Mipmap
- Mipmap size
- OFF/ON COMPARISON of texture mipmaps
- Bilinear, trilinear, anisotropic
- **Texture Display**
- Need of texture filtering
- Filtering types
- Nearest point sampling Vs Linear filtering
- 4 Texel weighted formula
- AMD's 7nm RDNA Radeon RX 5700 GPU
- **Texture mapping illustration**
- Texture space and its description
- Applying texture map
- Calculating scan extrema coordinates explanation

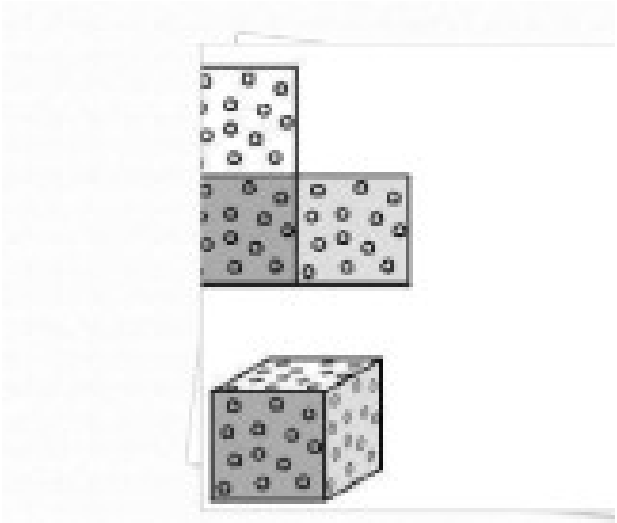
Texture mapping

- Texture mapping is a graphic design process in which a two dimensional surface called a texture map is wrapped around a three dimensional object.
- Thus the three dimensional object aquires a surface texture similar to that of the 2D surface.
- Texture mapping originally referred to diffuse mapping, a method that simply mapped pixels from a texture to a 3D surface ("wrapping" the image around the object).
- Mapping a two-dimensional texture onto a 3D model

Example

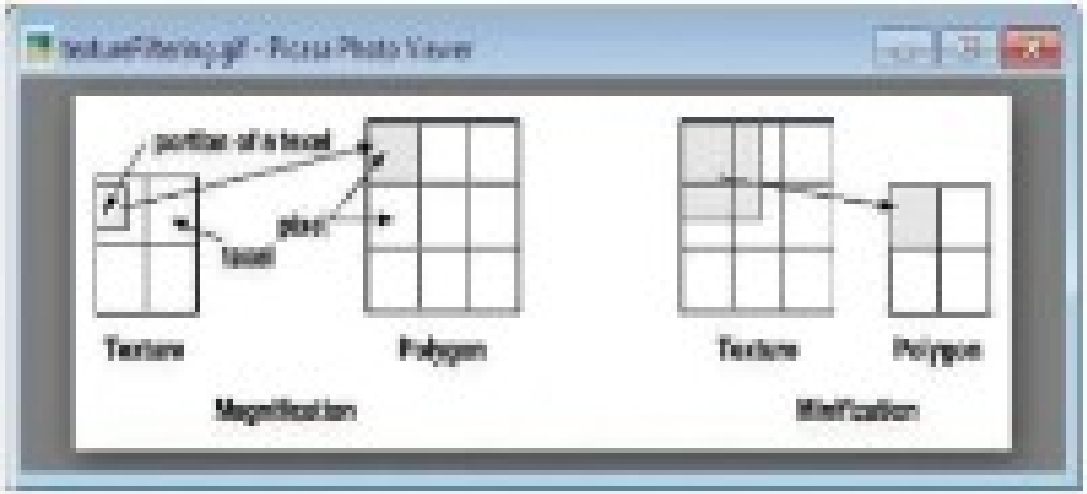


- Three identical squares each covered randomly with dots are directly mapped onto the three visible facets of a 3D cube.
- This distorts the size and shapes of the dots on the top and right hand facets.
- In this mapping the texture map covers the cube with no apparent discontinuities because of the way the dot are arranged on the squares.
- Mapping a 2D surface onto a 3D alters the size and shapes of the image elements.



Texel vs Pixel

- When a 3D texture mapped object appears close to the viewer so that the texture elements appear relatively large.
- There may be several pixels in each Texel the pattern of the texture map is easy to see.
- When the same 3D object is removed to increasing distances, the texture map pattern appears smaller and smaller.
- Eventually, each Texel can become smaller than a pixel. Then an averaging process must be used. Several Texel's are combined to form each pixel.
- If the object becomes distant enough, or if one of its facets appears at a sharp angle with respect to the viewer, the Texel may become so small that image.



The basics of texturing

- From left-to-right, we have the texture's *u,v* coordinates mapped directly to the corner vertices' *x,y* coordinates.
- Then the top vertices have had their *y* coordinates increased, but as the texture is still directly mapped to them, the texture gets stretched upwards.
- In the far right image, it's the texture that's altered this time: the *u* values have been raised but this results in the texture becoming squashed and then repeated.

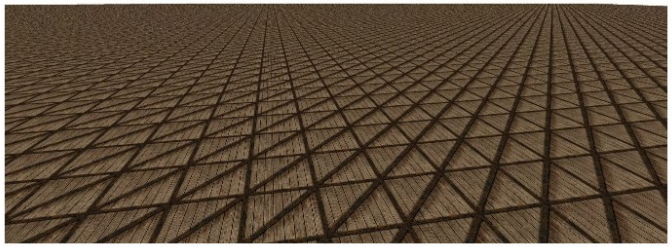


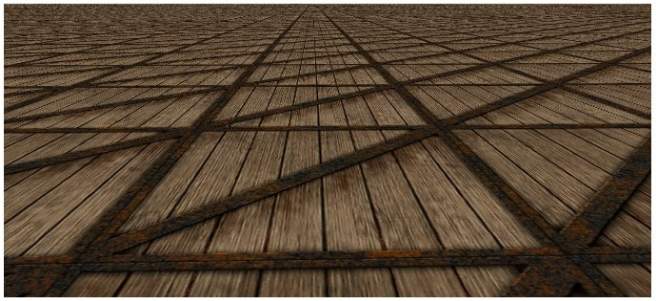
Texture repeating

- This is because although the texture is now effectively taller, the higher *u* value, it still has to fit into the primitive -- essentially the texture has been partially repeated.
- This is one way of doing something that's seen in lots of 3D games: t*exture repeating*.
- Common examples of this can be found in scenes with rocky or grassy landscapes, or brick walls.

Texture minification

- Now that crate texture, in its original gif format, is 66 kiB in size and has a resolution of 256 x 256 pixels.
- The original resolution of the portion of the frame that the crate textures cover is 1900 x 680, so in terms of just pixel 'area' that region should only be able to display 20 crate textures.
- We're obviously looking at way more than 20, so it must mean that a lot of the crate textures in the background must be *much* smaller than 256 x 256 pixels.
- Indeed they are, and they've undergone a process called *texture minification* (yes, that is a word!).
- Now let's try it again, but this time zoomed right into one of the crates.





Texture magnification

- Don't forget that the texture is just 256 x 256 pixels in size, but here we can see one texture being more than half the width of the 1900 pixels wide image.
- This texture has gone through something called *texture magnification*.
- These two texture processes occur in 3D games all the time, because as the camera moves about the scene or models move closer and further away, all of the textures applied to the primitives need to be scaled along with the polygons.
- Mathematically, this isn't a big deal, in fact, it's so simple that even the most basic of integrated graphics chips blitz through such work.
- However, texture minification and magnification present fresh problems that have to be resolved somehow.

Enter the mini-me of textures

- The first issue to be fixed is for textures in the distance.
- If we look back at that first crate landscape image, the ones right at the horizon are effectively only a few pixels in size.
- So trying to squash a 256 x 256 pixel image into such a small space is pointless for two reasons.
- One, a smaller texture will take up less memory space in a graphics card, which is handy for trying to fit into a small amount of cache.
- That means it is less likely to removed from the cache and so repeated use of that texture will gain the full performance benefit of data being in nearby memory.
- The second reason we'll come to in a moment, as it's tied to the same problem for textures zoomed in.

Mipmaps

- A common solution to the use of big textures being squashed into tiny primitives involves the use of *mipmaps*.
- These are scaled down versions of the original texture; they can be generated the game engine itself (by using the relevant API command to make them) or pre-made by the game designers.
- Each level of mipmap texture has half the linear dimensions of the previous one.
- So for the crate texture, it goes something like this: 256 x 256 → 128 x 128 → 64 x 64 → 32 x 32 → 16 x 16 → 8 x 8 → 4 x 4 → 2 x 2 → 1 x 1.

Mipmap texture having half linear dimensions



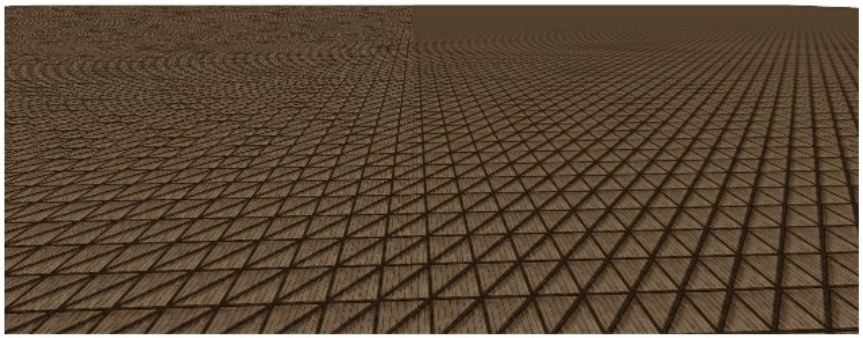
Determining Mipmap

- The mipmaps are all packed together, so that the texture is still the same filename but is now larger.
- The texture is packed in such a way that the *u,v* coordinates not only determine which texel gets applied to a pixel in the frame, but also from which mipmap.
- The programmers then code the renderer to determine the mipmap to be used based on the depth value of the frame pixel, i.e. if it is very high, then the pixel is in the far distance, so a tiny mipmap can be used.

Mipmap size

- Sharp eyed readers might have spotted a downside to mipmaps, though, and it comes at the cost of the textures being larger.
- The original crate texture is 256 x 256 pixels in size, but as you can see in the above image, the texture with mipmaps is now 384 x 256.
- Yes, there's lots of empty space, but no matter how you pack in the smaller textures, the overall increase to at least one of the texture's dimensions is 50%.
- But this is only true for pre-made mipmaps; if the game engine is programmed to generate them as required, then the increase is never more than 33% than the original texture size.
- So for a relatively small increase in memory for the texture mipmaps, you're gaining performance benefits and visual improvements.

- Below is is an **OFF/ON COMPARISON** of texture mipmaps:
- On the left hand side of the image, the crate textures are being used 'as is', resulting in a grainy appearance and so-called moiré patterns in the distance.
- Whereas on the right hand side, the use of mipmaps results in a much smoother transition across the landscape, where the crate texture blurs into a consistent color at the horizon.
- The thing is, though, who wants blurry textures spoiling the background of their favorite game?



Bilinear, trilinear, anisotropic

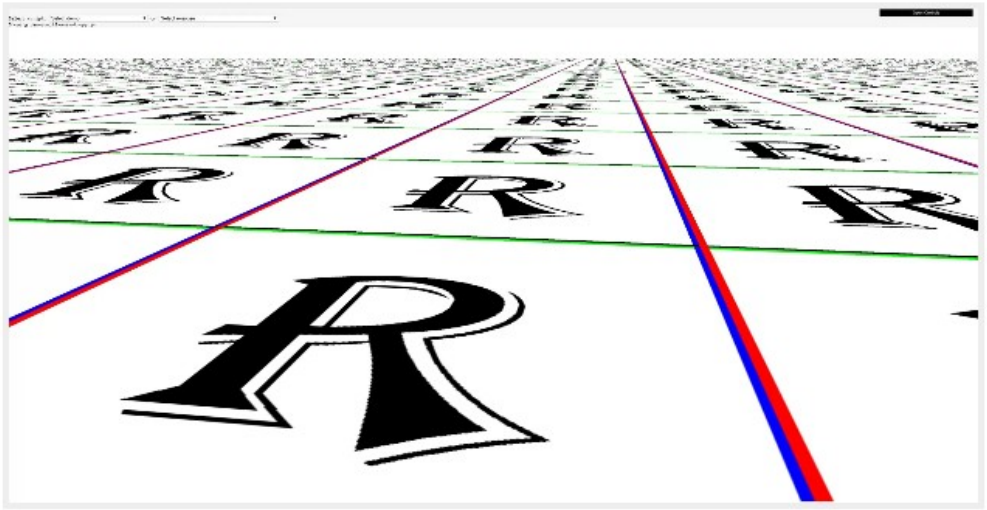
- The process of selecting a pixel from a texture, to be applied to a pixel in a frame, is called ***TEXTURE SAMPLING***, and in a perfect world, there would be a texture that exactly fits the primitive it's for -- regardless of its size, position, direction, and so on.
- In other words, texture sampling would be nothing more than a straight 1-to-1 texel-to-pixel mapping process.
- Since that isn't the case, texture sampling has to account for a number of factors:
  - Has the texture been magnified or minified?
  - Is the texture original or a mipmap?
  - What angle is the texture being displayed at?

Texture Display

- Let's analyze these one at a time.
- The first one is obvious enough: if the texture has been magnified, then there will be more texels covering the pixel in the primitive than required; with minification it will be the other way around, each texel now has to cover more than one pixel.
- That's a bit of a problem.
- The second one isn't though, as mipmaps are used to get around the texture sampling issue with primitives in the distance, so that just leaves textures at an angle.
- And yes, that's a problem too. Why?
- Because all textures are images generated for a view 'face on', or to be all math-like: the normal of a texture surface is the same as the normal of the surface that the texture is currently displayed on.

Need of texture filtering

- So having too few or too many texels, and having texels at an angle, require an additional process called *texture filtering*.
- If you don't use this process, then this is what you get:



Filtering types

- Here we've replaced the crate texture with a letter R texture, to show more clearly how much of a mess it can get without texture filtering!
- Graphics APIs such as Direct3D, OpenGL, and Vulkan all offer the same range filtering types but use different names for them.
- Essentially, though, they all go like this:
  - Nearest point sampling
  - Linear texture filtering
  - Anisotropic texture filtering

Nearest point sampling Vs Linear filtering

- To all intents and purposes, *nearest point sampling* isn't filtering - this is because all that happens is the nearest texel to the pixel requiring the texture is sampled (i.e. copied from memory) and then blended with the pixel's original color.
- Here comes linear filtering to the rescue.
- The required *u,v* coordinates for the texel are sent off to the hardware for sampling, but instead of taking the very nearest texel to those coordinates, the sampler takes *four* texels.
- These are directly above, below, left, and right of the one selected by using nearest point sampling.

4 Texel weighted formula

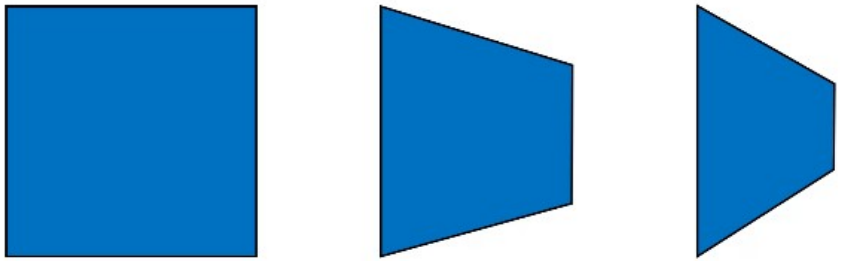
- These 4 texels are then blended together using a weighted formula. In Vulkan, for example, the formula is:

$$T_f = (1-\alpha)(1-\beta)T_1 + (\alpha)(1-\beta)T_2 + (1-\alpha)(\beta)T_3 + (\alpha)(\beta)T_4$$

- The *T* refers to texel color, where *f* is for the filtered one and 1 through to 4 are the four sampled texels.
- The values for *alpha* and *beta* come from how far away the point defined by the *u,v* coordinates is from the middle of the texture.

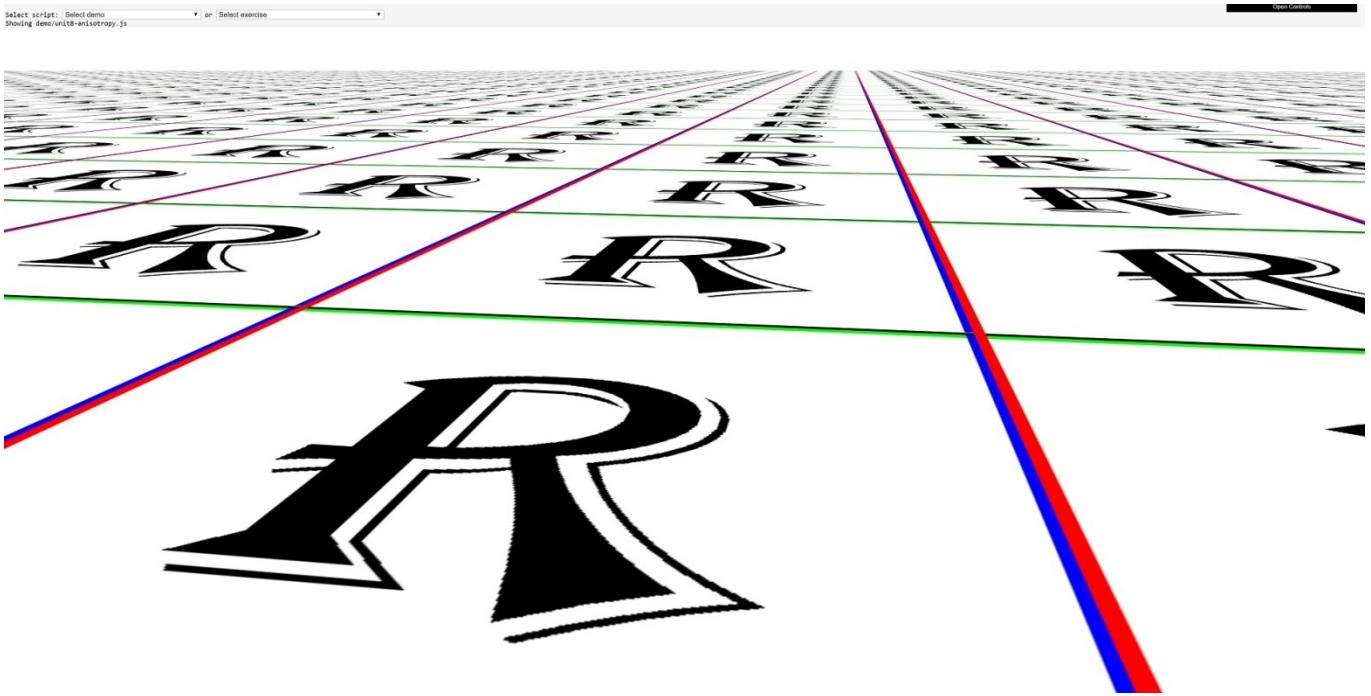
- Fortunately for everyone involved in 3D games, whether playing them or making them, this happens automatically in the graphics processing chip.
- In fact, this is what the TMU chip in the 3dfx Voodoo did: sampled 4 texels and then blended them together.
- Direct3D oddly calls this *bilinear filtering,* but since the time of Quake and the Voodoo's TMU chip, graphics cards have been able to do bilinear filtering in just one clock.

- Linear filtering can be used alongside mipmaps, and if you want to get really fancy with your filtering, you can take 4 texels from a texture, then another 4 from the next level of mipmap, and then blend all that lot together.
- The last filtering method to mention is called *anisotropic.*
- This is actually an adjustment to the process done in bilinear or trilinear filtering.
- It initially involves a calculation of the *degree of anisotropy* of the primitive's surface (and it's surprisingly complex, too) -- this value increases the primitive's aspect ratio alters due to its orientation:



- The above image shows the same square primitive, with equal length sides; but as it rotates away from our perspective, the square appears to become a rectangle, and its width increases over its height.
- So the primitive on the right has a larger degree of anisotropy than those left of it (and in the case of the square, the degree is exactly zero).
- Many of today's 3D games allow you to enable anisotropic filtering and then adjust the level of it (1x through to 16x), but what does that actually change?
- The setting controls the maximum number of additional texel samples that are taken per original linear sampling.
- For example, let's say the game is set to use 8x anisotropic bilinear filtering.
- This means that instead of just fetching 4 texels values, it will fetch 32 values.

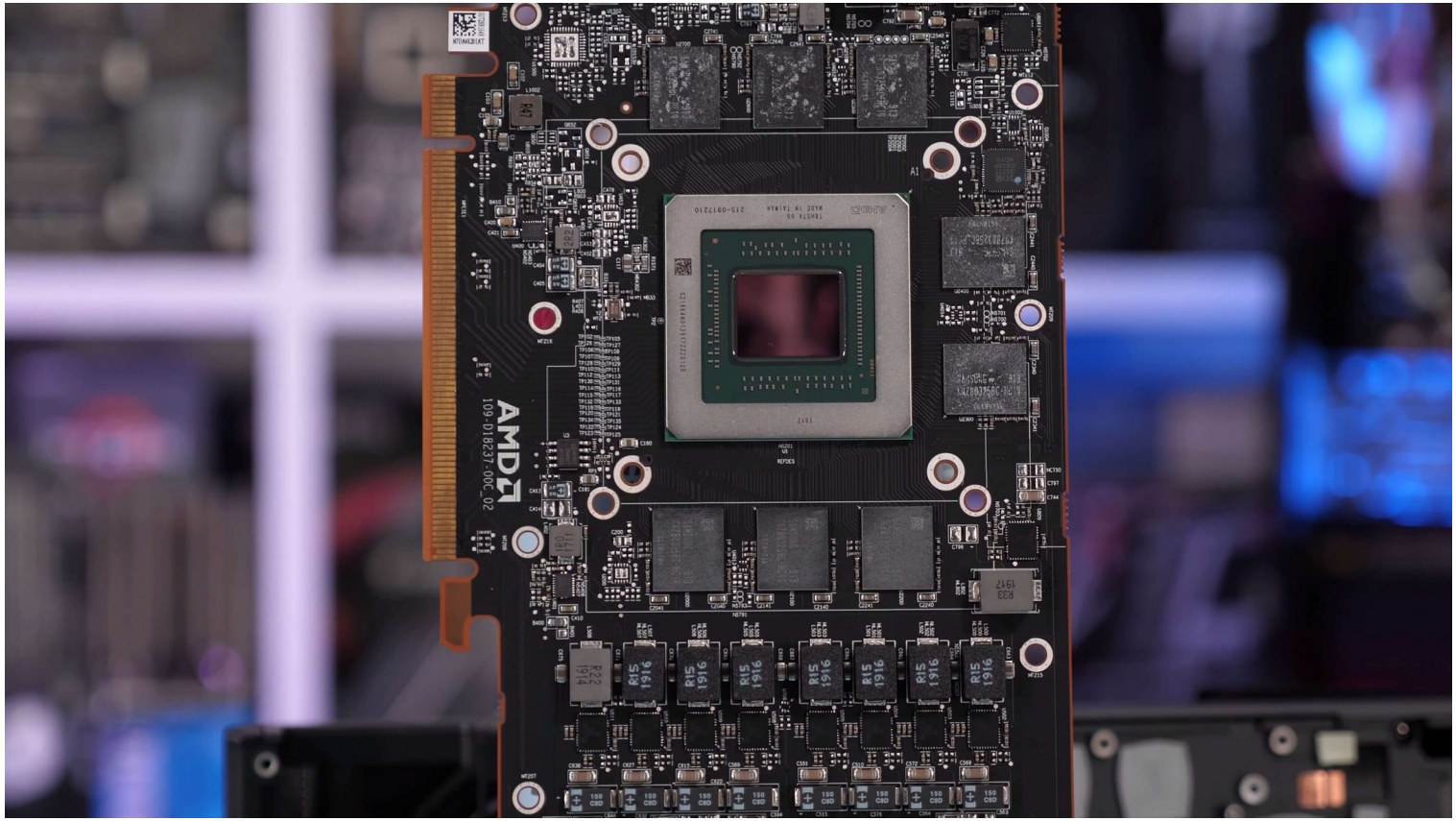
- The difference the use of anisotropic filtering can make is clear to see:



- compare nearest point sampling to maxed out 16x anisotropic trilinear filtering. So smooth.
- But there must be a price to pay for texture and it's surely performance: all maxed out, anisotropic trilinear filtering will be fetching 128 samples from a texture, for each pixel being rendered.
- For even the very best of the latest GPUs, that just can't be done in a single clock cycle.

- If we take something like AMD's Radeon RX 5700 XT, each one of the texturing units inside the processor can fire off 32 texel addresses in one clock cycle, then load 32 texel values from memory (each 32 bits in size) in another clock cycle, and then blend 4 of them together in one more tick.
- So, for 128 texel samples blended into one, that requires at least 16 clock cycles.

AMD's 7nm RDNA Radeon RX 5700 GPU



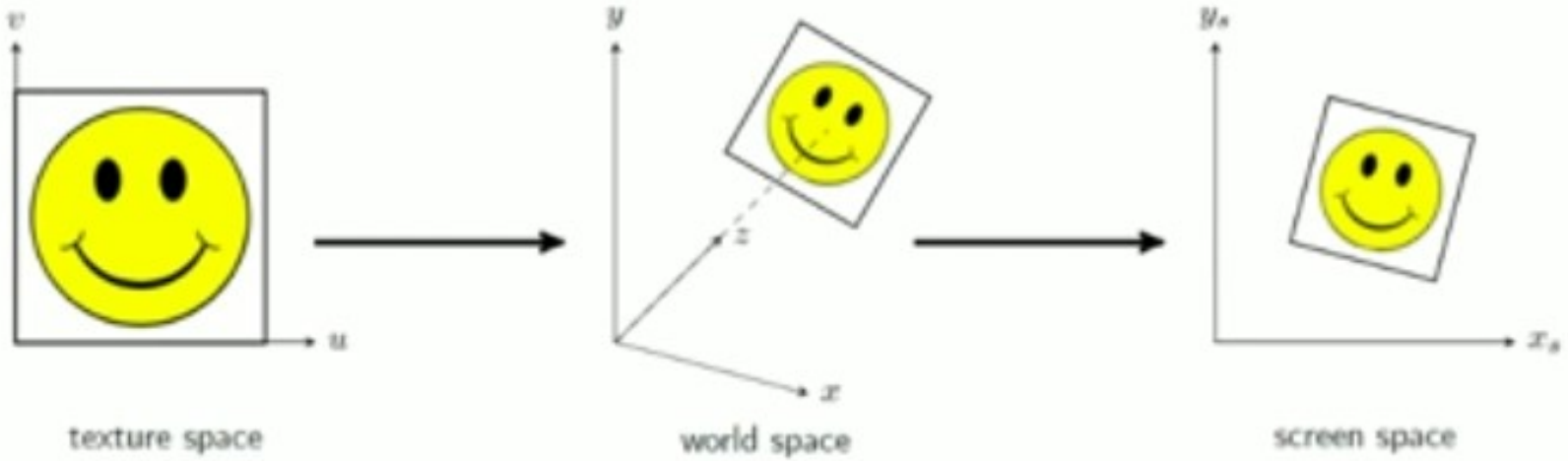
- Now the base clock rate of a 5700 XT is 1605 MHz, so sixteen cycles takes a mere 10 *nanoseconds*.
- Doing this for every pixel in a 4K frame, using just *one* texture unit, would still only take 70 milliseconds.
- 3Dfx Voodoo to handling textures could max out at 1 bilinear filtered texel per clock cycle, and with the TMU chip rocking along at 50 MHz, that meant 50 million texels could be churned out, every second. A game running at 800 x 600 and 30 fps, would only need 14 million bilinear filtered texels per second.
- However, this all assumes that the textures are in nearby memory and that only one texel is mapped to each pixel.
- Twenty years ago, the idea of needing to apply multiple textures to a primitive was almost completely alien, but it's commonplace now.

Colour determination

Texture Mapping

- **Texture mapping** is the process of mapping a 2D image onto a 3D polygon
- For example, a photograph of a brick wall can be mapped to a single polygon saving us having to use a polygon for each brick
- The image that is mapped onto the polygon is called a **texture**
- A pixel in the texture is called a **textel**
- The colours pixels in the raster are determined by the colours of the textels in the texture

Texture Mapping



Texture mapping illustration

- Texture space contains the object space.
- Object is then placed on the polygon in world space. Polygon is now in the projection plane.
- Bitmap, texture and vectors cannot be projected on the projection plane.
- Texture made up bitmap having pixels, a raster.
- Hence how to go from screen space to texture space is the problem.

Texture space

- The **texture space** is the space in which the texture exists
- It is a 2D space with horizontal and vertical axes denoted by $u \in [0, 1]$ and $v \in [0, 1]$
- A texture map fills the texture space, i.e., bottom-left corner is at $(0, 0)$ and top-right corner is at $(1, 1)$.
- The co-ordinates of a textel (U, V) for a $M_y \times M_x$ texture map are

$$U = \lfloor M_x \cdot u + 1 \rfloor,$$
$$V = \lfloor M_y \cdot v + 1 \rfloor.$$

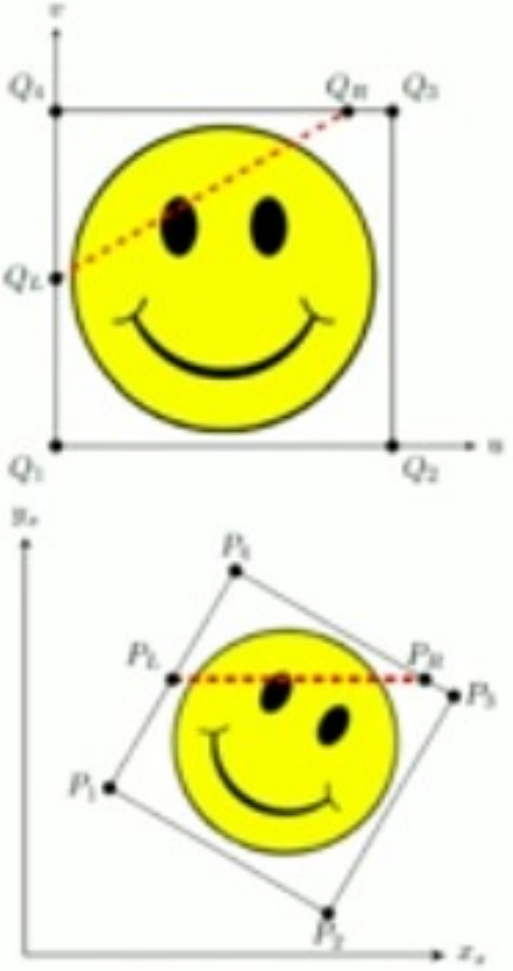
- When $u = 0$ then $U = 0$ and when $u = 1$ then $U = M_x + 1$, therefore we have to check to ensure $0 > u, v < 1$

Texture space description

- u and v are the texture coordinates. They range between 0 and 1.
- For example a photograph taken can be assumed between (0,0) and (1,1).
- Textels are texture-pixels which are discrete entities. Textel coordinates (U,V) have to be in a range 1 to the number of textels in the image.
- Mx is the number of textel in horizontal direction and My is the number of textel in vertical direction.
- Scaling is performed for U and V. u and v times the number of horizontal and vertical textels respectively in a floor $\lfloor \rfloor$ bracket representation).
- Floor means the integer should be round the number down to the nearest integer below.
- If the number of textel is 3.8, the number is round down to its nearest integer 3.
- This is how the textel coordinates are scaled. Hence U and V are introduced.

Applying a texture map

- The vertices of the polygon, P_i correspond to the vertices Q_i in the texture space
- Any transformation of the polygon should be accounted for by the texture mapping
- Using a scan line method, the scan extrema P_L is calculated by interpolating between P_4 and P_1
- Similarly the scan extrema P_R is calculated by interpolating between P_4 and P_3

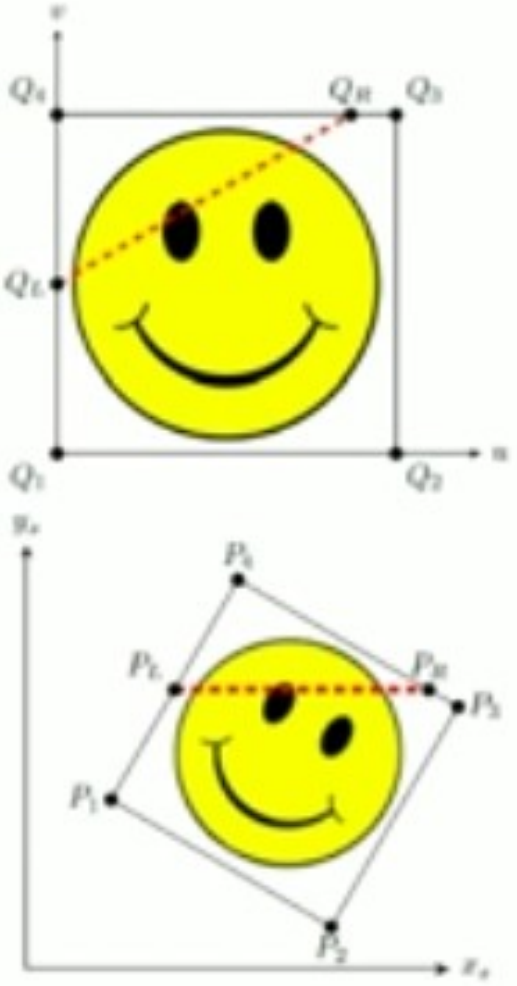


Applying a texture map

- The texture space range u is 0 to 1 and the same for v too.
- In the screen space, the polygon is deliberately rotated and is not parallel to the X and Y axis. Here the Xs and Ys are the screen space coordinates.
- In scanline method, horizontal rows of pixels are taken. PL and PR are the left and right scan extreme which is the left edge and the right edge of the polygon.
- The colours of the pixels on the scanline correspond to the dotted line on the texture space where the QL and QR correspond to PL and PR.
- The corners of the polygon P1,P2,P3,P4 in the screen space corresponds to the corners of texture Q1,Q2,Q3,Q4 in the texture space.

Applying a texture map

- The corresponding scan extrema points in the texture space Q_L and Q_R are calculated by interpolating along the edges of the texture
- Pixels between P_L and P_R take on the same colour as the textels between Q_L and Q_R



Calculating scan extrema co-ordinates

- The vertex co-ordinates are scaled to the size of the raster using

$$x = \max\left[1, (x_s + 1)\frac{N_x}{2}\right], \qquad y = \max\left[1, (y_s + 1)\frac{N_y}{2}\right].$$

- The scan extrema P_L and P_R are initialised to the vertex with the largest y_s co-ordinate (i.e., P_4)

- The y co-ordinate of P_L and P_R is calculated by subtracting 1 from the current y co-ordinate

- x_L is calculated by interpolating between P_4 and P_1

$$x_L = x_4 - \frac{y_4 - y}{y_4 - y_1}(x_4 - x_1)$$

Since $y = y_4 - 1$ then

$$x_L = x_4 - \frac{y_4 - y_4 + 1}{y_4 - y_1}(x_4 - x_1)$$

$$= x_4 - \frac{x_4 - x_1}{y_4 - y_1}$$

Calculating scan extrema coordinates explanation

- Consider screen space coordinates from -1 to +1.
- When clipping to unit cube between the coordinates from -1 to +1 therefore it has to be scaled which is a raster assignment.
- So the scaling is done by adding 1 to the screen coordinate Xs multiplied by the number of pixels in the screen divided by size of screen space from -1 to +1 is 2.
- In case of x, the max 1 ensures that if this is less than 1, it is not zero and there is a whole number coordinate. The similar case is in the y also.